Oracle® Solaris Studio 12.4: Code Analyzer User's Guide

Part No: E37090
October 2014

Contents

Using This Documentation

- **Overview** – Describes how to use the Code Analyzer tool, to analyze and display data
- **Audience** – Application developers, system developers, architects, support engineers
- **Required knowledge** – Programming experience, software development testing, experience in building and compiling software products

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at `http://docs.oracle.com/cd/E37069_01`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

Introduction

Oracle Solaris Studio Code Analyzer is an integrated set of tools that can help developers of C and C++ applications for Oracle Solaris produce secure, robust, and quality software.

This chapter includes information about the following:

Data Analyzed by The Code Analyzer

Code Analyzer analyzes three types of data:

- Static code errors detected during compilation
- Dynamic memory access errors and warnings detected by the `discover` utility, the memory error discovery tool
- Code coverage data measured by the `uncover` utility, the code coverage tool

In addition to providing you access to each individual type of analysis, Code Analyzer integrates static code checking with dynamic memory access analysis and code coverage analysis, to enable you to find many important errors in your applications that cannot be found by other error detection tools working separately.

The Code Analyzer also pinpoints the core issues in your code, that, when fixed, are likely to eliminate the other issues. A core issue usually combines several other issues because, for example, the issues have a common allocation point, or occur at the same data address in the same function.

Static Code Checking

Static code checking detects common programming errors in your code during compilation. The `-xprevise=yes` option for the C and C++ compilers leverages the compilers' control and data flow analysis frameworks to analyze your application for potential programming and security flaws.

Note - You can optionally use the `-xanalyze=code` option to collect static code errors, but this option is EOL. Using the `-xprevise=yes` option is recommended.

For information on collecting static error data, see "Collecting Static Error Data" on page 13.

For a list of the static code errors the Code Analyzer analyzes, see "Static Code Issues" on page 23.

Dynamic Memory Access Checking

Memory-related errors in your code are often difficult to find. When you instrument your program with `discover` before running it, `discover` catches and reports memory access errors dynamically during program execution. For example, if your program allocates an array and does not initialize it and then tries to read from a location in the array, the program is likely to behave erratically. If you instrument the program with Discover and then run it, `discover` will catch the error.

For information about collecting dynamic memory access error data, see "Collecting Dynamic Memory Access Data" on page 14.

For a list of the dynamic memory access issues that Code Analyzer analyzes, see "Dynamic Memory Access Errors" on page 28.

Code Coverage Checking

Code coverage provides information on which areas of your code are exercised in testing and which are not, enabling you to improve your test suites to test more of your code. Code Analyzer uses data collected by `uncover` to determine which functions in your program are uncovered and the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.

For information about collecting code coverage data, see "Collecting Code Coverage Data" on page 15.

Requirements for Using Code Analyzer

Code Analyzer works with static error data, dynamic memory access error data, and code coverage data collected from binaries compiled with the Oracle Solaris Studio 12.3 or 12.4 C or C++ compiler.

Code Analyzer runs on a SPARC-based or x86–based system running at least Solaris 10 10/08 operating system at least Oracle Solaris 11, Oracle Enterprise Linux 5.x, or Oracle Enterprise Linux 6.x.

Code Analyzer GUI

After collecting data with the compiler, Discover, or Uncover, you can start Code Analyzer GUI to display and analyze the issues by issuing the `code-analyzer` command.

For each issue, Code Analyzer displays the issue description, the path name of the source file in which the issue was found, and a code snippet from that file with the relevant source line highlighted.

Code Analyzer enables you to do the following:

- Display more details for an issue. For a static issue, the details include the Error Path. For a dynamic memory access issue, the details include a Call Stack and if the data is available, include an Allocation Stack and a Free Stack.
- Open the source file in which an issue was found.
- Move from a function call in the Error Path or stack to the associated source code line.
- Find all of the usages of a function in your program.
- Move to the declaration of a function.
- Move to the declaration of an overridden or overriding function.
- Display the call graph for a function.
- Display more information about each issue type, including a code example and possible causes.
- Filter the displayed issues by analysis type, issue type, and source file.
- Hide issues you have already reviewed, and close issues that you are not interested in.

For detailed information about using the GUI, see the online help in the GUI and “Oracle Solaris Studio 12.4: Code Analyzer Tutorial ”.

Code Analyzer Command-Line Interface

The command-line interface version of Code Analyzer, `codean`, reads the analytics file as input and generates output in text and HTML formats, using static code checking, Discover, and Uncover. It also provides a mechanism to store data in an history archive for later comparison of newer data with historic data. `codean` enables you to do the following:

- Read in the report in API format and transform the information into text and HTML format. `codean` saves text output to a `.`*type*`.html` file, where *type* can be either `static`, `dynamic`, or `coverage`.
- For the `.analyze/`*type*`/latest` report, calculate a checksum for each issue and store the original issue information in the `.analyze/history//`*type* file, where *type* can be either `static`, `dynamic`, or `coverage`.
- Show only the new or fixed issues in the latest report and compare it to previously saved reports.
- Specify what type of data to collect: `dynamic`, `static`, `coverage`, or `all`.
- Display the full path name.
- Display issues in specific source files.
- Display a certain number of lines from the source code.
- Save the latest reports.
- Overwrite the last saved report with the same tag name.
- Show only new or fixed issues in the report.
- Specify the directory in which to save your reports.
- Filter the types of errors and warnings to display.

For more information, see the codean(1) man page.

Remote Desktop Distribution

You can create a remote desktop distribution of Code Analyzer that will run on almost any operating system and use the Oracle Solaris Studio compilers and tools on a remote server. When you generate a remote desktop distribution during installation and check the Export User Settings From Default Directory option, Code Analyzer will recognize the server on which you generated the distribution as a remote host and access the tool collection in your Oracle Solaris Studio installation. This option is not checked by default.

To start the Code Analyzer on a remote operating system, run the appropriate executable:

```
./codeanalyzer/bin/codeanalyzer.exe
```

For information about how to install a Remote Desktop Distribution, see “Oracle Solaris Studio 12.4: Installation Guide ”.

For information about remote desktop distribution, see the Code Analyzer GUI online help.

Quick Start

The following is an example of the steps required to gather information about your code and how to view the results with Code Analyzer, using a sample C program.

▼ Quick Start

1. **Compile a program to collect static data.**

   ```
   % cc -xprevise=yes *.c
   ```

   Note - Previously, you could compile with the `-xanalyze=code` option. This option is still valid for Oracle Solaris Studio 12.4 but is EOL.

2. **Recompile program with debug information.**

   ```
   % cc -g *.c
   ```

3. **Instrument program with `discover` and run program to collect dynamic memory access data.**

   ```
   % cp a.out a.out.save
   % discover -a a.out
   % a.out
   ```

4. **Instrument program with `uncover` to collect code coverage data.**

   ```
   % a.out
   % cp a.out.save a.out
   % a.out
   % uncover a.out
   ```

5. **After the information has been gathered, you can choose to use Code Analyzer with the GUI or the `codean` command-line tool to display the collected data.**

   - **For accessing Code Analyzer with the GUI, use the following command:**

     ```
     % code-analyzer a.out
     ```

   - **For accessing Code Analyzer with the command-line tool, use the following command:**

```
% codean a.out
```

♦ ♦ ♦ CHAPTER 2

Collecting Data And Starting the Code Analyzer

The data you collect for analysis by the Code Analyzer is stored in the *binary-name*.analyze directory in the directory that contains your source code files. The *binary-name*.analyze directory is created by the compiler, discover, or uncover.

This chapter includes information about the following topics:

Collecting Static Error Data

To collect static error data on your C or C++ program, compile the program using Oracle Solaris Studio 12.3 or 12.4 C or C++ compiler with the -xprevise=yes option. Previously, you used the -xanalyze=code option, but this option is EOL and it is recommended to use the -xprevise=yes option instead. The -xprevise=yes option is not available in the compilers in previous releases of Oracle Solaris Studio. When you use this option, the compiler automatically extracts static errors and writes the data to the static subdirectory in the *binary-name*.analyze directory.

If you compile your program with the -xprevise=yes option and then link it in a separate step, you also need to include the -xanalyze=code option on the link step.

On Linux, you must specify the -xannotate option with -xprevise=yes in order to collect static error data. For example:

```
% cc -xprevise=yes -xannotate -g t.c
```

Note that the compilers cannot detect all of the static errors in your code.

- Some errors depend on data that is available only at runtime. For example, given the following code, the compiler would not detect an ABW (beyond array bounds write) error because it could not detect that the value of ix, read from a file, lies outside the range [0,9]:

```
void f(int fd, int array[10])
{
  int ix;
  read(fd, &ix, sizeof(ix));
  array[ix] = 0;
}
```

- Some errors are ambiguous,and also might not be actual errors. The compiler does not report these errors.
- Some complex errors are not detected by the compilers in this release.

After collecting static error data, you can start Code Analyzer's GUI or the command-line tool (`codean`) to analyze and display the data or recompile the program so that you can collect dynamic memory access or code coverage data.

Collecting Dynamic Memory Access Data

Collecting dynamic memory access data on your C or C++ program is a two-step process: instrumenting the binary with `discover` and then running the instrumented binary.

To instrument your program with `discover` to collect data for Code Analyzer, you must have compiled the program with Oracle Solaris Studio version 12.3 or 12.4 C or C++ compiler. Compiling with the `-g` option generates debug information that enables Code Analyzer to display source code and line number information for dynamic memory access errors and warnings.

`discover` provides the most complete detection of memory errors at the source code level if you compile your program without optimization. If you compile with optimization, some memory errors will not be detected.

For information about specific types of binaries that Discover can or cannot instrument, see "Prepare Binaries Correctly" in "Oracle Solaris Studio 12.4: Discover and Uncover User's Guide " and "Binaries That Use Preloading or Auditing Are Incompatible" in "Oracle Solaris Studio 12.4: Discover and Uncover User's Guide ".

Note - You can build your program once for use with both `discover` and `uncover`. However, because you cannot instrument a binary that is already instrumented, if you are also planning to use `uncover` to collect coverage data, save a copy of the binary for this purpose before instrumenting it with `discover`. For example:

```
% cp a.out a.out.save
```

▼ How to Collect Dynamic Memory Access Data From the Binary:

1. **Instrument the binary with Discover using the `-a` option:**

   ```
   % discover -a binary_name
   ```

   Note - You must use the version of Discover in Oracle Solaris Studio version 12.3 or 12.4. The `-a` option is not available in earlier versions of `discover`.

   ---

2. **Run the instrumented binary.**

   The dynamic memory access data is written to the `dynamic` subdirectory in the *binary_name*`.analyze` directory.

   Note - For additional instrumentation options you can specify when instrumenting the binary with `discover`, see "Instrumentation Options" in "Oracle Solaris Studio 12.4: Discover and Uncover User's Guide " or the `discover` man page.

   ---

3. **(Optional) Start Code Analyzer's GUI or the command-line tool (`codean`) to analyze and display the data, along with any static code data you might have previously collected. Or, you can use an uninstrumented copy of the binary to collect code coverage data.**

Collecting Code Coverage Data

Collecting code coverage data on your C or C++ program is a three-step process:

1. Instrumenting the binary with `uncover`
2. Running the instrumented binary
3. Running `uncover` again to generate a coverage report for use by Code Analyzer.

You can run the instrumented binary multiple times after instrumenting it, and accumulate data over all of the runs before generating the coverage report.

▼ How to Collect Code Coverage Data From the Binary

Before You Begin To instrument your program with `uncover` to collect data for use by Code Analyzer, you must have compiled the program with Oracle Solaris Studio version 12.3 or 12.4 C or C++ compiler.

Compiling with the `-g` option generates debug information that allows Code Analyzer to use source code level coverage information.

Note - If you saved a copy of the binary when you compiled your program for instrumenting with `discover`, you can rename the copy to the original binary name and use it for instrumenting with `uncover`. For example:

```
cp a.out.save a.out
```

---

1. **Instrument the binary with Uncover:**

   ```
   % uncover binary-name
   ```

2. **Run the instrumented binary one or more times.**

   The code coverage data is written to a *binary-name*`.uc` directory.

3. **Generate the code coverage report from the accumulated data using Uncover with the `-a` option:**

   ```
   % uncover -a  binary-name.uc
   ```

   The coverage report is written to the coverage subdirectory in the *binary-name*`.analyze` directory.

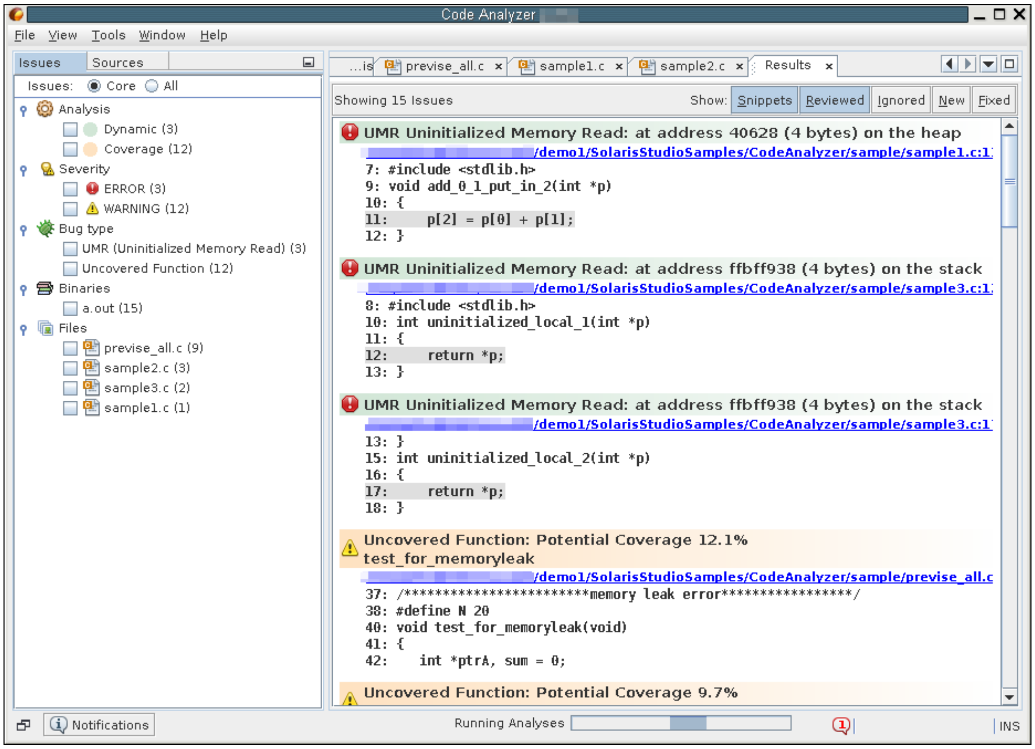
Note - You must use the version of `uncover` in Oracle Solaris Studio version 12.3 or 12.4. The `-a` option is not available in earlier versions of `uncover`.

Using the Code Analyzer GUI

You can use the Code Analyzer GUI to analyze up to three types of data. To start the GUI, type the `code-analyzer` command and the path to the binary for which you want to analyze error data you have collected:

```
% code-analyzer binary-name
```

The Code Analyzer GUI opens and displays the data in the *binary-name*`.analyze` directory, as shown in the following figure.


Code Analyzer [redacted]
File View Tools Window Help
Issues Sources
...is previse_all.c sample1.c sample2.c Results
Notifications
Running Analyses
INS

Issues: ● Core ○ All

- Analysis
  - □ Dynamic (3)
  - □ Coverage (12)
- Severity
  - □ ERROR (3)
  - □ WARNING (12)
- Bug type
  - □ UMR (Uninitialized Memory Read) (3)
  - □ Uncovered Function (12)
- Binaries
  - □ a.out (15)
- Files
  - □ previse_all.c (9)
  - □ sample2.c (3)
  - □ sample3.c (2)
  - □ sample1.c (1)

Showing 15 Issues

Show: [✓ Snippets] [✓ Reviewed] Ignored New Fixed

```
UMR Uninitialized Memory Read: at address 40628 (4 bytes) on the heap
  [redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c:1
    #include <stdlib.h>
    void add_0_1_put_in_2(int *p)
    {
        p[2] = p[0] + p[1];
    }

UMR Uninitialized Memory Read: at address ffbff938 (4 bytes) on the stack
  [redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample3.c:1
    #include <stdlib.h>
    int uninitialized_local_1(int *p)
    {
        return *p;
    }

UMR Uninitialized Memory Read: at address ffbff938 (4 bytes) on the stack
  [redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample3.c:1
    }
    int uninitialized_local_2(int *p)
    {
        return *p;
    }

Uncovered Function: Potential Coverage 12.1%
test_for_memoryleak
  [redacted]/demo1/SolarisStudioSamples/CodeAnalyzer/sample/previse_all.c
    /************************memory leak error*****************/
    #define N 20
    void test_for_memoryleak(void)
    {
       int *ptrA, sum = 0;

Uncovered Function: Potential Coverage 9.7%
```

When the Code Analyzer GUI is running, you can switch to displaying the data you have collected for a different binary by choosing Open → File and navigating to the binary.

The online help in the GUI describes how to use all of features to filter the displayed results, show or hide issues, and show more information about specific issues. The "Oracle Solaris Studio 12.4: Code Analyzer Tutorial " guides you through a complete scenario of data collection and analysis using a sample program.

Using the Code Analyzer Command-Line Tool (`codean`)

You can also use the Code Analyzer command-line tool `codean` to analyze up to three types of data. To start `codean`, type the `codean` command, any options, and the path of the executable or directory.

`codean` *options executable-path*|*directory*

The `codean` tool displays text output on the screen. You can also view the results in a `.`*type*`.html` file in the same place the executable resides. This section describes the command options

`codean` Options

The following sections explain the different options you can use for `codean`..

Data Type Options

The following options determine which type of data to collect.

`-s`	Process and display static data.
`-d`	Process and display dynamic data.
`-c`	Process and display coverage data.

You can specify multiple options or none. If none are selected, than the default is to process all possible options, depending on whether the `.analyze/`*type*`/latest` file exists, where *type* can be `static`, `dynamic`, or `coverage`.

Displaying Options

The following options determine the content of the text output of your results.

`--fullpath`	Display the full file's path name.
`-f` *source-file*	Display only the issues in the specified source file.
`-n` *number*	Display the specified number of lines of the source code.

Filtering Options

The following options determine the types of errors and warnings that are reported in the results.

The error or warning type can be one of the following:

- A three-letter error code or a three-letter warning code. For a list of possible errors and warnings, see Appendix A, “Errors Analyzed by Code Analyzer”.
- `MLK` or `mlk`, for memory leaks.

- `ALL` or `all`, for all warnings or errors.

If the error or warning is not specified, the default is `all`.

The filtering options are:

`--showerrors` *error-type*
Show only errors of the specified error type.

`--showwarnings` *warning-type*
Show only warnings of the specified warning type.

`--hideerrors` *error-type*
Do not show errors of the specified error type.

`--hidewarnings` *warning-type*
Do not show warnings of the specified warning type.

Saving Results Options

You can save your latest results in a file, placed in a specific directory with specific tag names.

`--save`
Save the latest reports.

`--tag` *tag-name*
When paired with `--save`, names the saved copy with the tag name *tag-name*. If a saved copy has the same tag name, `codean` issues a warning message and then exits without overwriting the file. If no tag name is specified, `codean` checks the last modified time of the latest report of the executable and uses the time stamp as the tag name.

`-t`
Overwrite the saved report with the same tag name.

`-D` *directory*
Save the report to the directory *directory*.

Comparing Results Options

The following options enable you to compare your results to a previously generated report.

`--whatisnew`
Show only new issues. This option cannot be used with `--whatisfixed`.

`--whatisfixed`
Show only fixed issues. This option cannot be used with `--whatisnew`.

`--tag` *tag-name* When paired with `--whatisnew` or `--whatisfixed`, uses the historic copy of the report with tag name *tag-name* to compare against newly generated report. If no tag name is specified, the latest report is compared against the last saved copy.

`--ref` *file|directory* Must be paired with `--whatisnew` or `--whatisfixed` and must have a path name following it. This option specifies which file or directory to compare the new report against.

codean Work Flow Example

This section provides an example of monitoring the effect of a bug fix.

1. Compile the target source before the fix.

   ```
   % cc -g *.c
   ```

2. Instrument the binary using Discover and make sure it generates Analytics output.

   ```
   % discover -a a.out
   ```

3. Run the instrumented binary.
4. Use `codean` to store the analytics output. The history archive is created at `a.out.analyze/history/before_bugfix` and a history file called `dynamic` is created in this directory.

   ```
   % codean --save --tag before_bugfix -d a.out
   ```

5. Fix the bug.
6. Compile the target source again.

   ```
   % cc -g *.c
   ```

7. Instrument the binary again using `discover`.

   ```
   % discover -a a.out
   ```

8. Run the instrumented binary.

   ```
   % a.out
   ```

9. Show comparison results and ensure that the invalid memory access caused by the bug is fixed.

   ```
   % codean --whatisfixed --tag before_bugfix -d a.out
   ```

   This produces a new Analytics output file at `a.out.analyze/dynamic/fixed_before_bugfix` and that contains only fixed dynamic issues. You can use `codean` or the Code Analyzer GUI to view these fixed issues.

10. (Optional) Run `codean` to ensure you did not introduce any new bugs.

    ```
    % codean --whatisnew --tag before_bugfix -d a.out
    ```

This command produces a new analytics file at a.out.analyze/dynamic/new_before_bugfix that contains only new dynamic issues.

♦ ♦ ♦ APPENDIX A

Errors Analyzed by Code Analyzer

The compilers, `discover`, and `uncover` find static code issues, dynamic memory access issues, and coverage issues in your code. This appendix describes the specific error types that are found by these tools and analyzed by Code Analyzer.

- "Code Coverage Issues" on page 23
- "Static Code Issues" on page 23
- "Dynamic Memory Access Errors" on page 28
- "Dynamic Memory Access Warnings" on page 35

Code Coverage Issues

Code coverage checking determines which functions are uncovered. In the results, code coverage issues found are labeled as Uncovered Function, with a potential coverage percentage, indicating the percentage of coverage that will be added to the total coverage for the application if a test covering the relevant function is added.

Possible Causes: No test might execute your function or you might have forgotten to delete dead or old code.

Static Code Issues

Static code checking finds the following types of errors:

- ABR: beyond array bounds read
- ABW: beyond array bounds write
- DFM: double freeing memory
- ECV: explicit type cast violation
- FMR: freed memory read
- FMW: freed memory write
- INF: infinite empty loop
- MLK: memory leak

- MFR: missing function return
- MRC: missing malloc return value check
- NFR: uninitialized function return
- NUL: null pointer dereference, leaky pointer check
- RFM: return freed memory
- UMR: uninitialized memory read, uninitialized memory read bit operation
- URV: unused return value
- VES: out-of-scope local variable usage

This section describes possible causes of the error and a code example of when the error might occur.

Beyond Array Bounds Read (ABR)

Possible causes: Attempting to read memory beyond the array bounds.

Example:

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

Beyond Array Bounds Write (ABW)

Possible causes: Attempting to write memory beyond the array bounds.

Example:

```
int a [5];
 . . .
 a[5] = 5; // Writing to memory beyond array bounds
```

Double Freeing Memory (DFM)

Possible Causes: Calling `free()()` more than once with the same pointer. In C++, using the `delete` operator more than once on the same pointer.

Example:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .      // p was not signed a new value between the free statements
 free(p); // Double freeing memory
```

Freed Memory Read (FMR)

Example:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . .  // Nothing assigned to p in between
printf("p = 0x%h\n",p); // Reading from freed memory
```

Freed Memory Write (FMW)

Example:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .        // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

Infinite Empty Loop (INF)

Example:

```
int x=0;
int i=0;
while (i200) {
  x++; } // Infinite loop
```

Memory Leak

Possible causes: Memory is allocated but not freed before exit or escaping from the function.

Example:

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                      // The 2nd malloc leaked here
```

Missing Function Return (MFR)

Possible causes: Missing return values along some paths to exit.

Example:

```
#include <stdio.h>
int foo (int a, int b)
{
  if (a)
    {
       return b;
    }
}            // If foo returns here, the return is uninitialized
int main ( )
{
    printf("%d\n", foo(0,30));
}
```

Missing Malloc Return Value Check (MRC)

Possible causes: Accessing a return value from `malloc` in C or a new operator in C++ without checking against `null`.

Example:

```
#include <stdlib.h>
int main()
{
 int *p3 = (int*) malloc(sizeof(int)); // Missing null-pointer check after malloc.
 *p3 = 0;
}
```

Leaky Pointer Checker: Null Pointer Dereference (NUL)

Possible causes: Accessing a pointer that might equal to `null`, or redundant checking against `null` in case the pointer is never null.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int gp, ctl;
int main()
{
 int *p = gp;
 if (ctl)
  p = 0;
 printf ("%c\n", *p); // May be null pointer dereference
 if (!p)
```

```
  *p = 0; // Surely null pointer dereference

int *p2 = gp;
*p2 = 0; // Access before checking against NULL.
assert (p2!=0);

int *p3 = gp;
if (p3) {
  printf ("p3 is not zero.\n");
}
*p3 = 0; // Access is not protected by previous check against NULL.
}
```

Return Freed Memory (RFM)

Example:

```
#include <stdlib.h>
int *foo ()
{
 int *p = (int*) malloc(sizeof(int));
 free(p);
 return p; // Return freed memory is dangerous
}
int main()
{
 int *p = foo();
 *p = 0;
}
```

Uninitialized Memory Read (UMR)

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
#include <stdio.h>
#include <stdlib.h>
 struct ttt {
    int a: 1;
    int b: 1;
 };

 int main()
 {
   int *p = (int*) malloc(sizeof(int));
   printf("*p = %d\n",*p); // Accessing uninitialized data

   struct ttt t;
```

```
  extern void foo (struct ttt *);

  t.a = 1;
  foo (&t); // Access uninitialized bitfield data "t.b"
}
```

Unused Return Value (URV)

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
 int foo();
int main()
{
   foo(); // Return value is not used.
}
```

Out-of-Scope Local Variable Usage (VES)

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
 int main()
{
   int *p = (int *)0;
   void bar (int *);
   {
     int a[10];
     p = a;
   } // local variable 'a' leaked out
   bar(p);
 }
```

Dynamic Memory Access Errors

Dynamic memory access checking finds the following types of errors:

- ABR: beyond array bounds read
- ABW: beyond array bounds write
- BFM: bad free memory
- BRP: bad realloc address parameter
- CGB: corrupted guard block

- DFM: double freeing memory
- FMR: freed memory read
- FMW: freed memory write
- FRP: freed realloc parameter
- IMR: invalid memory read
- IMW: invalid memory write
- MLK: memory leak
- OLP: overlapping source and destination
- PIR: partially initialized read
- SBR: beyond stack bounds read
- SBW: beyond stack bounds write
- UAR: unallocated memory read
- UAW: unallocated memory write
- UMR: uninitialized memory read

This sections describes the possible causes of the error and a code example of when the error would occur.

Beyond Array Bounds Read (ABR)

Possible causes: Attempting to read memory beyond the array bounds.

Example:

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

Beyond Array Bounds Write (ABW)

Possible causes: Attempting to write memory beyond the array bounds.

Example:

```
int a [5];
 . . .
 a[5] = 5; // Writing to memory beyond array bounds
```

Bad Free Memory (BFM)

Possible Causes: Passing a non-heap data pointer to `free()()` or `realloc()()`.

Example:

```
#include <stdlib.h>
int main()
{
 int *p = (int*) malloc(sizeof(int));
 free(p+1); // Freeing wrong memory block
}
```

Bad Realloc Address Parameter (BRP)

Example:

```
#include <stdlib.h>
int main()
{
 int *p = (int*) realloc(0,sizeof(int));
 int *q = (int*) realloc(p+20,sizeof(int[2])); // Bad address parameter for realloc
}
```

Corrupted Guard Block (CGB)

Possible Causes: Writing past the end of a dynamically allocated array, or being in the "red zone".

Example:

```
   #include <stdio.h>
   #include <stdlib.h>

   int main() {
    int *p = (int *) malloc(sizeof(int)*4);
    *(p+5) = 10; //  Corrupted array guard block detected (only when the code is not
annotated)
    free(p);

    return 0;
   }
```

Double Freeing Memory (DFM)

Possible Causes: Calling `free()()` more than once with the same pointer. In C++, using the `delete` operator more than once on the same pointer.

Example:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .      // p was not assigned a new value between the free statements
 free(p); // Double freeing memory
```

Freed Memory Read (FMR)

Example:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .  // Nothing assigned to p in between
 printf("p = 0x%h\n",p); // Reading from freed memory
```

Freed Memory Write (FMW)

Example:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .       // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

Freed Realloc Parameter (FRP)

Example:

```
#include <stdlib.h>

int main() {
  int *p = (int *) malloc(sizeof(int));
  free(0);
  int *q = (int*) realloc(p,sizeof(it[2])); //Freed pointer passed to realloc
}
```

Invalid Memory Read (IMR)

Possible causes: Reading 2, 4, or 8 bytes from an address that is not half-word aligned, word aligned, or double-word aligned, respectively.

Example:

```
#include <stdlib.h>
```

```
int main()
{
  int *p = 0;
  int i = *p;   // Read from invalid memory address
}
```

Invalid Memory Write (IMW)

Possible causes: Writing 2, 4, or 8 bytes from an address that is not half-word aligned, word aligned, or double-word aligned, respectively. Writing to a text address, writing to a read-only data section (`.rodata`), or writing to a page that `mmap` has made read-only.

Example:

```
  int main()
{
  int *p = 0;
  *p = 1;  // Write to invalid memory address
}
```

Memory Leak

Possible causes: Memory is allocated but not freed before exit or escaping from the function.

Example:

```
    int foo()
{
  int *p = (int*) malloc(sizeof(int));
  if (x) {
   p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                      // The 2nd malloc leaked here
```

Overlapping Source and Destination (OLP)

Possible causes: Incorrect source, destination, or length is specified. When the source and destination overlap, the behavior of the program is undefined.

Example:

```
#include <stlib.h>
#include <string.h>
int main() {
  char *s=(char *) malloc(15);
```

```
   memset(s, 'x', 15);
   memcpy(s, s+5, 10);
   return 0;
 }
```

Partially Initialized Read (PIR)

Example:

```
 #include <stdio.h>
#include <stdlib.h>
 int main()
{
  int *p = (int*) malloc(sizeof(int));
  *((char*)p) = 'c';
  printf("*(p = %d\n",*(p+1)); // Accessing partially initialized data
}
```

Beyond Stack Bounds Read (SBR)

Possible causes: Reading a local array past the end or before the start.

Example:

```
 #include <stdio.h>

int main() {
  int a[2] = {0, 1};
  printf("a[-10]=%d\n",a[-10]); //  Read is beyond stack frame bounds

  return 0;
}
```

Beyond Stack Bounds Write (SBW)

Possible causes: Writing to a local array past the end or before the start.

Example:

```
 #include <stdio.h>

 int main() {
  int a[2] = {0, 1};
  a[-10] = 2; //  Write is beyond stack frame bounds

  return 0;
```

```
}
```

Unallocated Memory Read (UAR)

Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:

```
 #include <stdio.h>
 #include <stdlib>
int main()
{
  int *p = (int*) malloc(sizeof(int));
  printf("*(p+1) = %d\n",*(p+1)); // Reading from unallocated memory
}
```

Unallocated Memory Write (UAW)

Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:

```
  #include <stdio.h>
  #include <stdlib>
 int main()
 {
  int *p = (int*) malloc(sizeof(int));
   *(p+1) = 1; // Writing to unallocated memory
}
```

Uninitialized Memory Read (UMR)

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
  #include <stdio.h>
  #include <stdlib>
 int main()
 {
  int *p = (int*) malloc(sizeof(int));
  printf("*p = %d\n",*p); // Accessing uninitialized data
}
```

Dynamic Memory Access Warnings

Dynamic memory access checking finds the following types of warnings:

- AZS: allocating zero size
- Memory leak
- SMR: speculative uninitialized memory read

This section describes the possible causes of the warning and a code example of when the warning might occur.

Allocating Zero Size (AZS)

Example:

```
#include <stdlib>
int main()
{
  int *p = malloc(); // Allocating zero size memory block
}
```

Memory Leak (MLK)

Possible causes: Memory is allocated but not freed before exit or escaping from the function.

Example:

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
 }
}                                     // The 2nd malloc leaked here
```

Speculative Memory Read (SMR)

Example:

```
int i;
if (foo(&i) != 0)  /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
```

The compiler might generate the following equivalent code for the above source:

```
    int i;
    int t1, t2'
    t1 = foo(&i);
    t2 = i; /* value in i is loaded. So even if t1 is 0, we have uninitialized read due to
speculative load */
    if (t1 != 0)
    printf("%d\n", t2);
```

Index